Importing and transforming data: some basic techniques

Statistics for Social Scientists II

Bur, GJM

2024-09-14

1 Importing data

Write a set of commands that import the three data-sets from my Github. You should download these manually rather than directly call them from my Github. Here is what this should look like in pseudo-code. *See my lab syllabus for exact format.*

```
as comments: date, name, task
as code:
start a log
change your directory
clear existing data from memory
use the data I posted
```

Don't worry if it seems a little silly to clear, load, and then again clear and load. We're going to add commands in a moment, but it made sense to work on loading all sets first.

2 Transforming data: basic

As Gordon correctly notes, "We can always recreate the modified data file in a future session by re-running the batch program that created it" (ignore "batch program"; this just means a do-file).

*You should always transform your data with a **script** or **do-file**, and you should generally **avoid** saving it—there is no point unless your cleaning takes a lot of time.* At this level, the code should run quickly unless your data-set is several hundred megabytes or you do a massive amount of cleaning. The downside of saving your edits to the data as a new data-set is that it is *extremely* easy to forget how you got that set from the publicly-available set, and now your research is unreproducible—not good.

Here are some basic transformations. By the way, in what follows, a dollar sign in a code-block (i.e., something like this: `sum $yourvar$`) just means "your input goes between the dollar signs".

1. In the GSS data, make a variable called `retirement_age` that indicates whether someone is old enough to collect Social Security or not. There are many ways to do this. The fastest is `gen $newvar$ = $condition$`, e.g. `gen in20s = (age > 19 & age <`

`30)`. Check your work with `vers 16: table $dummyvar$, c(min $continuousvar$ max $continuousvar$)` (the versioning is useful because Stata changed the `table` syntax to something clunkier for some reason).

2. Make a value label and a variable label for your new variable. To make a value label, you can use the following syntax.

```
label define $labelname$ 0 "$labelfor0$ 1 "$labelforone$" 2 "$labelfortwo$" // etc.
label values $newvariable$ $labelname$
```

3. In the CPS extract, make a variable that indicates whether someone is married, called simply `married`. To do this correctly, you'll need to view the value label for the old variable, which you can also use to make a value label for the new variable. Do so as follows:

```
describe $oldvar$
// look at the name of the label listed
label list $oldvarvaluelabel$
```

Check you work with a cross-tab, rather than a table of means (since we're just going from categorical to dummy); for each row of your new dummy variable, your matrix should only have non-zero entries in one column. **Why?**.

```
tab $oldvar$ $newvar$
```

Once you've done this correctly, make a new value label for your dummy.

4. In the same data-set, check for implausible values of hours worked last week. Drop those observations using the conditional syntax listed above.

5. Use the Detroit Area Study data-set. Make a value label for the marital variable (you may need to search the codebook to work this out, or you can search in the variable window). Make sure to assign missings that *are currently* given real values to `.`, which is Stata's default missing value.

3 Transforming data: advanced

6. Make a set of dummy variables that indicate that a variable is an outlier on various continuous variables in the CPS *and* standardize each variable: `hourslw`, `age`, `wage1`. You can do this manually or with a loop.

In general, a loop is basically very much like a sum; indeed, we could manually calculate means and variances with a loop (computationally this is inefficient, but it's useful to see).

Note that the command `local` stores information in a temporary variable in Stata; this disappears after a given run of a do-file (if you are working interactively, they persist until the end of a session), so you need to run all of this in one go. You can access the contents of a local by referring to it with the "left" and "right" tickmarks that you see below (no need to use them when defining the local).

`forvalues` is just Stata's way of saying "what are the lower and upper indices of a sum" (in most languages, this is reduced to `for`), and `mpg[`j']` accesses the jth element of

the column `mpg`. Note that the braces and indentation are necessary. Finally, why do I re-define the variable inside the loop? This is a conventional way of updating the sum with each run through of the loop (other languages make this automatic, but it's still a good trick to no).

```
local samptotal = 0
local sumsq = 0
local n = 74
forvalues j = 1/`n' {
    local samptotal = `samptotal' +  mpg[`j']
}
di `samptotal'/`n'
local ybar = `samptotal'/`n'

local sumsq = 0
forvalues j = 1/`n' {
    local sumsq = `sumsq' +  (mpg[`j'] - `ybar')^2
}

di `sumsq'/(`n'-1)

sum mpg, d
```

OK, that was a bit tricky. As a freebie, here is the loop syntax for this specific problem; you just need to change a very small amount. But make sure you understand what I'm doing (this uses the Tukey rule for outliers).

````
```
foreach var of varlist $var1$ $var2$ $etc...$ {
    quietly sum `var', d
    local q3 = r(p75)
    local q1 = r(p25)
    local ybar = r(mean)
    local s = r(sd)
    local iqr = `q3' - `q1'
    gen $outlier_`var'$ = (`var' >`q3' + 1.5*`iqr') | (`var' <`q1' - 1.5*`iqr')
    gen $z_`var'$ = (`var' - `ybar')/`s'
    }
```
````